CS422 - Final Exam

Time: until noon, Tuesday Dec 12. Send it either by email or push it under my door (SC 2110). Total: 100 (out of 110) points needed for maximum credit. You are *not* allowed to collaborate or discuss these problems with each other. You can use books, lecture notes, computers, coffee, etc.

Problem 1. (15 points)
Consider programs of the following form, written in a `FUN`-like language supporting the various parameter passing styles discussed in class:

```
let f(P x, P y) = x * y
and g(Q x, Q y) = {
                      x := x + y ;
                      y := x - y ;
                      x := x - y ;
                      x + y
                  }
and x = a and y = b
in f(g(x,y), g(y,y))
```

Here `a` and `b` are some arbitrary integers, assumed given. `P` and `Q` stay for parameter passing styles. What values do these programs evaluate to, when `P` is any of call-by-value, call-by-name, or call-by-need, and when `Q` is any of call-by-value and call-by-reference? For each of the six situations, explain how you obtained the result.

Problem 2. (20 points)
Suppose that you are the designer of a concurrent language and that you'd like to add support for transactional actions into your language by means of an atomicity construct. More precisely, you'd like to add a statement atomic{E}, which evaluates E atomically, that is, without any interference from the other threads (assume that you have already added thread creation and termination to your language). Also, you'd like to add a statement break-atomic, which leaves the atomic block discarding all the computation together with its side effects that have been accumulated while partially executing the atomic block, and unfreezes the rest of the threads. If the atomic block terminates normally, then its side-effects are committed. In other words, atomic{E} freezes the other threads and starts to evaluate E optimistically; if E evaluates normally, then its effects are committed, the other threads are unfrozen, and the computation continues normally; if the evaluation of E encounters a break-atomic statement, then the state of the program is recovered to the moment in which the atomic block was tried as if none of its code has been executed, the atomic block is skipped, the other threads are unfrozen, and the computation continues normally, as if the atomic block was never seen.

This problem has two parts:

1. Sketch K definitions for atomic{E} and break-atomic;
2. Comment on the computational limitations of your definition above, namely on the fact that the rest of the world is frozen in order for an atomic block to stay atomic. Can you do better?

Can you let the other threads continue their executions and stop them only if they try to interfere with with the atomic block?

Problem 3. (20 points)
Suppose that you have the following object-oriented program in a language combining OO and functional features:

```
class A {
  field value;
  method initialize(v) { value := v }
  method m() { value + 10 }
}
class B inherits A {
  method m() { super m() }
}
class C inherits B {
  field obj;
  method void initialize(v,o) {
    super initialize(v);
    obj := o
  }
  method m() { value + send obj m() }
}
class D inherits B {
  method initialize(v) { super initialize(v) }
  method m() { value * 2 }
}
main
  let b = ...
  in let o = if b then new C(5, new C(10, new D(15))) else new D(20)
     in send o m()
```

Translate and run this program in KOOL. What value will be returned under dynamic method invocation when b is true? When b is false? Recall that KOOL is a dynamically dispatched language, like Java. Under static method dispatch, one would statically analyze the program and find precisely which method is being invoked by each call. The main advantage of static dispatch is its efficiency: one needs no runtime search. C++ has static method dispatch by default, but also provides support for dynamic method dispatch (using virtual methods). Comment on what you'd need to do to transform KOOL into a statically dispatched language. Would you need to take typing into account? How about conditionals? What values would the program above evaluate to under static method dispatch, when b is true and when b is false?

Problem 4. (35 points)

(10 points) Write a FUN program that calculates all the k-combinations of n elements, that is, all the possibilities to pick k elements from a pool of n elements. It is known that there are $n!/(k! \cdot$

$(n-k)!$) such combinations; you are supposed to define a function taking two arguments n and k and returning a list of lists of numbers between 1 and n; these lists are regarded as sets (no repetitions and the order of elements does not matter).

(10 points) Type the program above by hand using the type inference technique discussed in class. Give enough detail to show that you understand the technique, but not more than that.

(15 points) Apply the CPS transformation to the program above by hand using the technique discussed in class. Again, give enough detail to show that you understand the CPS transformation technique.

Problem 5. (20 points)

Write the dining philosophers program in FUN (with concurrency). You can assume just 3 philosophers and you can assume that each acquires first the left fork, then the right one, then eat (don't need to do anything here, just put a comment in the code saying "eating"), then release the right fork and then the left one. The philosophers must be threads and the forks must be locks. Say how many states this 3-threaded program can possibly reach, where states are counted only when rewrite rules apply (the only rewrite rule that you need here is the one for acquiring a lock). You are allowed to use Maude's search capabilities to count the number of states, though you do not need or have to. How can you say when the program is deadlocked? How many deadlock states are there? Is ordinary testing powerful enough to detect this deadlock?